Two Fundamental Concepts in Skeletal Parallel Programming

Two Fundamental Concepts in Skeletal Parallel Programming

Anne Benoit and Murray Cole

School of Informatics, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
{abenoit1, mic}@inf.ed.ac.uk
http://homepages.inf.ed.ac.uk/mic/Skeletons

Abstract. We define the concepts of *nesting mode* and *interaction mode* as they arise in the description of skeletal parallel programming systems. We suggest that these new concepts encapsulate fundamental design issues and may play a useful role in defining and distinguishing between the capabilities of competing systems. We present the decisions taken in our own Edinburgh Skeleton Library *eSkel*, and review the approaches chosen by a selection of other skeleton libraries.

1 Introduction

The skeletal approach to parallel programming is well documented in the research literature (see [1, 2, 3, 4] for surveys). It observes that many parallel algorithms can be characterised and classified by their adherence to one or more of a number of generic patterns of computation and interaction. For instance, a variety of applications in image and signal processing are naturally expressed as process pipelines, with parallelism both between pipeline stages, and within each stage by replication and/or more traditional data parallelism [5].

Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations but implementations which recognise these to enhance performance. This level of abstraction makes it easier for the disciplined programmer to experiment with a variety of parallel structurings for a given application, by enabling a clean separation between structural aspects and the application specific details. Meanwhile, the explicit structural information provided by the use of skeletons enables static and dynamic optimisations of the underlying implementation.

In the *eSkel* (Edinburgh Skeleton Library) project [6, 7, 8], motivated by our observations [2] on previous attempts to implement these ideas, we have begun to define a generic set of skeletons as a library of C functions on top of MPI. This work has caused us to identify two issues which we believe to be fundamental to the definition of any skeletal parallel programming system, but which have been addressed only implicitly in previous work.

The main contribution of this paper, in Section 2, is to present these underlying concepts explicitly. Section 3 outlines the approach we take in *eSkel*,

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 764–771, 2005.

comparing this in Section 4 with the main characteristics of a number of previous projects. Our conclusions are in Section 5.

2 Fundamental Concepts

The concepts highlighted by this paper may usefully be labelled *nesting mode* and *interaction mode*. In order to present these clearly and concisely, we begin by introducing some generic terminology for other important aspects of skeletal programming. We will then introduce the new concepts. In doing so, we will provide simple examples, which involve multiple uses of only one actual skeleton, the pipeline. This simplification allows us to focus precisely on the underlying issues, which might be obscured by the use of further skeletons and real applications.

2.1 Skeleton Terminology

The purpose of any skeleton, irrespective of the language framework within which it is embedded, is to abstract a pattern of *activities* and their *interactions* (where the activities might be processes or threads, and the interactions might be message based communications or exchanges through shared variables). Activities might themselves be internally parallel.

It is helpful to further distinguish *internal* and *external* interactions. *Internal* interactions occur between two or more activities, whereas *external interactions* occur between activities and the enclosing context. For example, in a pipeline, an interaction between two stages is internal, but an interaction between the first stage and the source of pipeline inputs (which might be a data buffer, or some other thread or process, or even another skeleton) is external.

The definition of a particular skeleton constrains the ways in which its constituent activities may interact. In general such constraints have two aspects. For a given activity, *spatial constraints* determine the activities with which it *may* interact and the directions these interactions may take. We will call these the *partner activities*. For example, in a pipeline, activities correspond to stages and for a given stage its partners are the preceding and succeeding stages (where these exist), with the data restricted to flow *from* the predecessor *to* the successor. Meanwhile, *temporal constraints* determine the allowable orderings of interactions between partners. For example, in a particularly strict form of pipeline, we may require that a stage interacts first with its predecessor then its successor, in strict alternation.

2.2 Fundamental Concept: Nesting Mode

Conceptually, skeletons may be nested in two ways. In the discussion that follows we will refer to *outer* and *inner* levels to indicate what might be respectively thought of as the "calling" and "called" skeleton instances in a nesting. In a multiple level nesting, all but the outermost and innermost skeleton instances will play both roles.

Transient Nesting. In a **transient nesting**, an activity may decide to invoke another skeleton in order to process some locally available data, or to perform some other self-contained computation. For example, a pipeline stage with several internal processes may receive an input datum, partition it into a collection of sub-data, and process these independently with an internal pipeline, before reconstituting a single outer level result datum from the collected inner level results. This is then transmitted to the subsequent outer level stage. Each invocation of the nested pipeline is initiated by the outer level activity, between, and so independently of, interactions at the outer level. A transient nesting is therefore really no more than a conventional function call, albeit one invoked concurrently by all members of the outer level activity.

Persistent Nesting. In contrast, in a **persistent nesting** an activity at the outer level handles a complete sequence of enclosing level interactions in a skeletally structured fashion. From the perspective of the inner skeleton, its external interactions become bound to interactions (which may be internal or external depending upon context) at the outer level. For example, a pipeline stage (at the outer level) could choose to invoke a persistent pipeline as its inner level. Each request for a datum by the first stage of the inner level would be (conceptually at least) mapped to a request for a datum by the enclosing activity (ie the outer level pipeline stage). The inner skeleton invocation persists for the duration of the outer skeleton invocation. In particular it persists across outer level interactions, and indeed *subsumes responsibility* for dealing with these.

Example & Discussion. We now illustrate the nesting modes with a simple example. The example is deliberately artificial, since our intention is to contrast the essence of the modes without application specific clutter. For the same reason, we use only the pipeline skeleton, since we wish to focus on the interaction and nesting mechanisms, rather than the particular patterns encapsulated.

Consider a four stage pipeline, in which each stage (outer level activity) is assigned three processors. The pipeline processes 100 inputs, where each input is a block of 10 integers. Thus, each outer level activity performs 100 input and 100 output interactions. Now consider some possible nested structures for the second stage.

We could choose to design a second stage as follows. Upon receipt of each new input, the datum is decomposed into a sequence of 10 individual integers, which are then passed through a three stage inner level pipeline formed by the processors responsible for this outer level stage. Having processed the ten integers, the inner pipeline is dissolved (its call terminates) and the ten results are reconstituted into a single outer level datum, for forwarding to the third stage at the outer level. The process repeats with the next outer level datum. For each invocation of the inner level pipeline, each processor within the second outer level stage processes a stream of 10 inputs in its role as an inner level pipeline stage. Since there will be 100 such invocations, each processor processes 1000 inputs (each a single integer) in total. This is a transient nesting.

In a different application but with the same data types and structure at the outer level, the second outer level stage might choose to use its three processors

as a nested persistent pipeline. In this case, a request for a new datum by the first stage at the inner level corresponds directly to a request for an input datum by the second stage at the outer level. The nested skeleton is invoked once for the entire computation. Now each of the three second (outer) stage processors processes 100 inputs (each a block of 10 integers) *in total* during its role as an inner level pipeline stage. It is interesting to note that in this artificial example, the structure is operationally equivalent to that which could have been obtained by expressing the structure as a single level, six (NB not seven) stage pipeline. This is entirely natural, and serves to highlight the distinction between persistent and transient nesting (since this equivalence does not hold in the transient case).

2.3 Fundamental Concept: Interaction Mode

The concept of *interaction mode* concerns the extent to which a skeleton constrains temporal, as well as spatial interaction. In general, it is reasonable to consider that this property may vary from one invocation of the same skeleton to another, and even between the constituent activities of one invocation. Thus, interaction mode is a property of an activity.

Implicit Interactions. An activity using **implicit interaction mode** makes interactions which are constrained both spatially and temporally by the skeleton.

Explicit Interactions. An activity using **explicit interaction mode** makes interactions which are constrained spatially by the skeleton but which are unconstrained temporally, being triggered explicitly instead by actions in the activity code.

Examples & Discussion. Consider a stage in an image processing pipeline, which is producing one transformed output image for every input image. We might think of this as the "normal" behaviour for a pipeline. It can be completely encapsulated by the constraints of the *implicit* mode for a pipeline stage activity. Another pipeline might contain a stage activity which is behaving as a filter: some inputs are accepted and passed on (perhaps after processing) to the subsequent stage, while others are removed from the stream. In this case some explicit action is required of the activity to trigger each input interaction. Consequently the activity has *explicit* interaction mode. Finally, imagine a stage which is generating several output interactions for each input interaction (or indeed without any input interactions at all, as occurs for example in the "generator" stage of the well known "prime sieve" pipeline). Again, some *explicit* trigger would be required to indicate availability of a new output. In general, we can imagine that a fully flexible skeleton should allow the programmer to specify the interaction mode for each activity.

3 Nesting and Interaction in *eSkel*

The *eSkel* library [8] provides the C/MPI programmer with a set of skeletal collective operations. We now explain how version 2 of the library addresses the

fundamental issues discussed in the previous section. As before, to avoid clutter we will not discuss eSkel's specific collection of skeletons, since these are orthogonal to the handling of modes. Activities in *eSkel* are defined as C functions, to be called in SPMD style by all processes participating in the activity, and passed to the skeleton call as function pointers.

Nesting Mode. *eSkel* allows both transient and persistent nesting in order to maximize flexibility. In concrete terms, the nesting mode is actually expressed through what *eSkel* calls the "data mode" parameter to a skeleton call. This determines whether the data which will be processed by the skeleton invocation is present in a (possibly distributed) buffer, provided by the calling processes as a further parameter to the call (so called "buffer mode" data) or that it will flow into the skeleton from the activities (and indirectly, the buffer) of some enclosing skeleton call (so called "stream mode" data). Thus, a skeleton called in buffer mode effectively carries or creates its own data, and so is *transiently* nested with respect to any enclosing skeleton call. In contrast, a skeleton called with stream data mode is by necessity (and implication) *persistently* nested. As an aside, we note that the outermost skeleton call in any *eSkel* program must have buffer mode data (because there is nowhere else for data to come from), and so is effectively transiently nested within any enclosing (non-skeletal) code.

Interaction Mode. In *eSkel*, a distinct interaction mode is associated with each activity in a skeleton instantiation by setting corresponding parameters to the skeleton call. For non-nested skeletons this involves a simple choice between the two standard modes, *implicit* or *explicit*, as discussed above. In the case of nested skeletons, we allow a further level of flexibility, in that a skeleton call may devolve its choice of interaction mode for any of its activities to be determined by a further skeleton, nested within that activity's code. This is achieved by specifying that the outer level activity has *devolved* activity mode. This process can be repeated arbitrarily many times within a hierarchy of nestings, to be ultimately resolved by the "leaf" activities, which must have one of the two fundamental interaction modes.

An implicit mode activity has no control over its interactions, so is described in *eSkel* as a function from input data items to output data items. In general, a single interaction may involve several inputs and/or several outputs, to various partner activities (for example, consider the "stencil" interaction at the heart of a typical two dimensional iterative relaxation algorithm). The implicit mode activity function is invoked every time a new collection of inputs arrives (under the control of the library implementation of the skeleton, which is taking care of the interactions). The semantics of the interactions are specified in terms of **eDM**, the *eSkel Data Model*. The unit of transfer during the interactions is an *eDM molecule*, which consists of a collection of *eDM atoms* (each containing the data relating to a distinct partner activity). The type of the data is defined using standard MPI datatypes. Full details of the eSkel data model are presented in [2].

By contrast, an activity with explicit interaction mode is invoked only once per skeleton call. Its interactions are triggered directly in the activity code, by

calling the functions `Take` and `Give`. These have a generic interface, but semantics defined for each skeleton (embodying the skeleton's spatial constraints). For example, calling `Give` in an explicit mode pipeline stage indicates that the data given as a parameter is ready for transfer to the successor stage.

4 The Fundamental Concepts in Other Skeleton Libraries

We now briefly introduce a selection of existing skeleton languages and libraries and consider their handling of nesting and interaction. The results are summarized in the following table, which should be read in conjunction with the discussion below, particularly for the cases annotated with *.

Library	Interaction mode		Nesting mode	
	Explicit	Implicit	Persistent	Transient
P3L	No	Yes	Yes	No
Lithium	No	Yes	Yes	No*
ASSIST	Yes	Yes	No*	No*
Kuchen	Yes*	Yes	Yes	Yes*
Eden	Yes	Yes*	Yes*	Yes*

P3L. The Pisa Parallel Programming Language (P3L) [3, 9] was designed to help the design of parallel applications by providing skeletons as new language constructs (rather than a library) which co-ordinate fragments of sequential code written in C. The **Anacleto** [10] compiler for P3L generates C and MPI. P3L inspired industrial collaboration, leading to **SkIE** [11], a heterogeneous environment for high performance computing applications. SkIE includes a graphical editor, allowing fast and intuitive design of the parallel components of an application. Interaction within P3L skeletons is via streams of data, which are interfaced to the sequential code through parameter lists and manipulated through ordinary C types and operations. No choice is given for the interaction mode - all interactions are implicitly defined by the skeleton. All nesting of skeletons is persistent, being defined within the P3L layer, which is syntactically separated from the sequential code defining the activities. Transient nesting is therefore not possible.

P3L-based Libraries. One of the first attempts to provide a skeleton programming environment via a plain C library was achieved with **SKElib** [12], which was based on the P3L conceptual (but not syntactic) model. Similarly, **Lithium** [13] is a more recently developed library which takes a corresponding approach on top of Java. These libraries inherit their interaction modes from P3L. In Lithium, skeletons and skeleton nestings are constructed as objects which are then executed by invoking a `parDo` method. This form of nesting is persistent. The concept of transient nesting is not explicitly addressed, and

while not syntactically forbidden, it appears that the current implementation does not support it.

ASSIST. More recently, the programming environment ASSIST [14] (A Software development System based upon Integrated Skeleton Technology) is being developed in Pisa. It allows more flexibility than the original P3L approaches. It proposes a generic skeleton, *parmod*, which has few constraints but includes the classical P3L style skeletons. The interactions can be either implicit and constrained by a skeleton, or can be explicitly defined by the user. No nesting of skeletons is allowed - complex structures must be expressed directly through the flexibility afforded by *parmod*.

Kuchen's Skeleton Library. Herbert Kuchen has proposed a skeleton library [15] on top of C++. This library proposes polymorphic higher-order functions which are efficiently implemented in parallel. It offers data parallel and task parallel skeletons, and a two level model in which data parallel skeletons may be invoked from within task parallel skeleton nests. Most interactions have implicit mode, but explicit mode is also supported for task parallel skeletons. Nesting of task parallel skeletons is persistent since, like Lithium, the task parallel process topology is fixed at construction. The data parallel skeletons allow more flexibility with transient nesting performed via direct calls to the functions.

Eden. The parallel functional language Eden [16] provides a model which is explicit about processes and their incoming and outgoing data, but abstracts from the detailed transfers and synchronisation. An Eden program defines a system of processes exchanging data on communication channels which are visible to the programmer as lazy lists. Explicit interactions are invoked by explicit manipulations of these lists. A number of skeletons have been defined on top of the Eden model [4, 17]. The examples presented leave the programmer with explicit control of interaction, but it would be simple to define further skeletons which abstract to implicit mode. Nesting issues are not addressed but it is clear that transient nesting can be achieved by calling a skeleton function at any time in the inner code of a skeleton, while persistent nesting can be achieved by appropriate use of Eden's process composition.

5 Summary and Conclusions

We have introduced the concepts of *nesting mode* and *interaction mode* as aspects of the design of skeletal parallel programming frameworks. We have examined several existing skeletal systems and have observed that design decisions related to these concepts are often made implicitly, and perhaps even as side-effects of other design decisions. We argue that these issues should be addressed explicitly in a design, and in its specification. We believe that the conceptual basis established here will help future designers to clarify and justify their decisions. We are attempting to follow this philosophy in the ongoing design and development of the *eSkel* library.

References

1. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press & Pitman, ISBN 0-262-53086-4 (1989)
2. Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing **30** (2004) 389–406
3. Pelagatti, S.: Structured Development of Parallel Programs. Taylor & Francis, London (1998)
4. Rabhi, F., Gorlatch, S., eds.: Patterns and Skeletons for Parallel and Distributed Computing. Springer Verlag, ISBN 1-85233-506-8 (2003)
5. Subhlok, J., O'Hallaron, D., Gross, T., Dinda, P., Webb, J.: Communication and memory requirements as the basis for mapping task and data parallel programs. In: Proceedings of Supercomputing '94, Washington, DC (1994) 330–339
6. Cole, M.: eSkel: The **e**dinburgh **Ske**leton **l**ibrary. Tutorial Introduction. Internal Paper, School of Informatics, University of Edinburgh (2002)
7. Cole, M.: eSkel: The **e**dinburgh **Ske**leton **l**ibrary Version 2.0 – Draft API reference manual. Internal Paper, School of Informatics, University of Edinburgh (2003)
8. Benoit, A., Cole, M.: eSkel's web page. (2005) `http://homepages.inf.ed.ac.uk/mic/eSkel`.
9. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P3L: A structured high level programming language and its structured support. Concurrency: Practice and Experience **7** (1995) 225–255
10. Ciarpaglini, S., Danelutto, M., Folchi, L., Manconi, C., Pelagatti, S.: ANACLETO: a template-based P3L compiler. In: Proceedings of the PCW'97, Camberra,Australia (1997)
11. Bacci, B., Danelutto, M., Pelagatti, S., Vanneschi, M.: SkIE: a heterogeneous environment for HPC applications. Parallel Computing **25** (1999) 1827–1852
12. Danelutto, M., Stigliani, M.: SKElib: parallel programming with skeletons in C. In: Proceedings of EuroPar2000. Number 1900 in LNCS, Springer-Verlag (2000)
13. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Generation Computer Systems **19** (2003) 611–626
14. Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M., Zoccolo, C.: ASSIST as a Research Framework for High-performance Grid Programming Environments. In Cunha, J.C., Rana, O.F., eds.: Grid Computing: Software environments and Tools, Springer Verlag (2004)
15. Kuchen, H.: A skeleton library. In: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Springer-Verlag (2002) 620–629
16. Breitinger, S., Loogen, R., Ortega-Mallén, Y., Peña, R.: Eden: Language Definition and Operational Semantics. Technical Report 10, Philipps-University of Marburg (1996)
17. Galán, L.A., Pareja, C., Pena, R.: Functional Skeletons Generate Process Topologies in Eden. In: Proc. of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, Springer-Verlag (1996) 289–303